An efficient string matching algorithm with k differences for nucleotide and amino acid sequences

Gad M.Landau, Uzi Vishkin[1] and Ruth Nussinov[2]*

Department of Computer Science, School of Mathematical Sciences, Tel Aviv University, Tel Aviv, Israel, [1]Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer St., New York, NY 10012, USA and [2]Sackler Institute of Molecular Medicine, Sackler Faculty of Medicine, Tel Aviv University, Ramat Aviv 69978, Israel

Received 8 July 1985

ABSTRACT

There are a few algorithms designed to solve the problem of the optimal alignment of one sequence, the *pattern*, of length m, with another, longer sequence the *text*, of length n. These algorithms allow mismatches, deletions and insertions. Algorithms to date run in $O(mn)$ time. Let us define an integer, k, which is the maximal number of differences allowed. We present a simple algorithm showing that sequences can be optimally aligned in $O(k^2n)$ time. For long sequences the gain factor over the currently used algorithms is very large.

1. Introduction

The problem of searching for homologies between sequences has been repeatedly addressed during the last few years. With the explosive growth of nucleotide (and amino acid) sequence data, there has been an urgent need for efficient algorithms which would compare the sequences. Although here we shall focus on the nucleotide sequence comparisons, it is eviden' that the same algorithms can be used to compare protein sequences as well.

Our algorithm (1,2) is most efficient in cases where we want to compare one sequence, the "pattern", of length m to another sequence, the "text" of length n, and n is much larger than m. Such situations arise often when a sequence is much longer than the pattern sequence, or, if we align the pattern with several (or many) other sequences, whose combined lengths would then be n. This happens when we want to find the site of the optimal alignment between a section from one sequence and another, or when we compare one gene with several related genes, or, with possible candidates for pseudogenes. We may also wish to compare the degree of potential relatedness between introns, or repetitive sequences.

* We have a preliminary version of the program and are currently working on an advanced one. When complete, it will be available for distribution upon mailing a request and tape to the first author.

A simple approach to finding good alignments was devised by Tinoco et al (3) and by Maizel and Lenk (4). It constructs an $m \times n$ matrix and scores matches of minimal lengths. The resulting diagonals are then visually inspected. For homologous sequences an elongated, nearly consecutive set of diagonals is observed. Often, however, the viewer is uncertain which set of diagonals to choose. Needleman and Wunsch (5) devised an elegant algorithm for finding an optimal alignment for which the number of matches as penalized by insertion/deletion gaps is maximal. Sankoff (6), Sellers (7) and Dumas and Ninio (8) have further extended this algorithm by minimizing the measure of sequence discrepancy or by maximizing the measure of similarity.

Korn et al (9) have written a program for finding good local alignments. Sellers (10, 11) has also treated the local alignment problem and devised an algorithm for finding within any pair of sequences all locally-close subsequences. Goad and Kanehisa (12) have introduced into his procedure a better measure of the quality of the detected alignments. A weight has been assigned to each deletion or mismatch type and a search for all subsequences in which the density of the differences is less than a previously set value is carried out. The Goad-Kanehisa program is probably the most sensitive method to locate all local homology alignments between two sequences. A much faster, approximate procedure has been devised by Wilbur and Lipman (13). Nussinov has presented an efficient code searching for global DNA homology (14) which is based on a maximal matching algorithm (15, 16). Fickett (17) has greatly improved the Seller's algorithm by disregarding computationally irrelevant steps.

To date, the algorithms used run in $O(mn)$ time. We define an additional parameter, k. k is the largest number of differences (i.e. mismatches, insertions, or deletions) allowed in the alignments of the pattern and the text. The algorithm presented here uses $O(k^2n)$ time. In addition to the large economy in time, the memory requirements of this algorithm are greatly reduced as well. Whereas in the classical method (3-17) the memory requirement is $O(mn)$, here the required memory is $O(m^2 + k^2)$.

We shall next describe two algorithms for finding the minimal number of differences between two strings, proceed to outline the new algorithm and end with a review of the biological implications of such algorithms.

2. An Algorithm for Finding The Minimal Number of Differences Between Two Strings

Suppose we have two strings (i.e. nucleotide sequences): $R = r_1r_2 \cdots r_m$ and $B = b_1b_2 \cdots b_n$. In our case each string is composed of an alphabet of four letters, A,C,G,T. On each of the strings there are three types of possible editing operations (i.e. mutations):

(i) deleting a character from position i to yield $R = r_1 r_2 \cdots r_{i-1} r_{i+1} \cdots r_m$;

(ii) inserting a character z to yield $R = r_1 r_2 \cdots r_i z r_{i+1} \cdots r_m$;

(iii) changing one character to yield $R = r_1 r_2 \cdots r_{i-1} z r_{i+1} \cdots r_m$. These editing operations have been penalized. The problem is, then, what is the minimal cost of editing operations that will transform sequence R into sequence B. This cost is denoted $D(R,B)$.

To simplify the presentation each difference between the two sequences is assigned a penalty of one unit, i.e. $C_{CH} = C_{DE} = C_{IN} = 1$, where C_{CH}, C_{DE} and C_{IN} are the costs of changing, deleting or inserting nucleotides. Now the cost of transforming sequence R into sequence B is simply the count of the minimal number of differences involved.

In matrix D sequence R is stretched along the y axis ($i = 1, \ldots, m$) and sequence B ($j = 1, \ldots, n$) is along the x-axis. The $D_{i,j}$ entry contains the attempted matching of $r_1 \cdots r_i$ and $b_1 \cdots b_j$. Say, we want to further inspect entries in the matrix. We can continue either by walking along a row ($i \rightarrow i + 1$), by walking along a column ($j \rightarrow j + 1$), or by increasing both parameters at the same time ($i \rightarrow i + 1, j \rightarrow j + 1$) and sliding along a diagonal d.

Each of the above steps has biological implications. Progressing along a row implies that the text (sequence B) has some nucleotides which the pattern (sequence R) lacks. That is, sequence B contains an insertion. Walking along a column in the D matrix implies that B lacks some nucleotides which are present in R, i.e., there is a deletion in the text. Continuation along a diagonal involves considering longer sequences with either a match or a mismatch.

It is clear that advancing along a diagonal consisting of elements $D_{i,j}$ with $j - i = d$ is preferable to advancing along a row or a column. Consider nucleotides r_i and b_j, with $r_i \neq b_j$. Continuing along only one sequence, say B, we may at best, match the previously unmatched last nucleotide of B. That is, $r_{i+1} = b_j$ and r_i is looped out of the alignment. Suppose we continue further along the column and consider now the pair r_{i+2}, b_j. For the purpose of obtaining an optimal alignment of R and B it is immaterial whether r_{i+2} and b_j match or not. If $r_{i+2} \neq b_j$, then the previous alignment is kept and r_{i+2} dangles at the end. If $r_{i+2} = b_j$, then we may keep this match, but now in addition to $r_{.}$ r_{i+1} is looped out as well. This state of affairs is bound to continue. Either the loop grows and/or the unmatched dangling end grows. The same situation holds for going along a column, i.e. keeping r_i constant and increasing b. The entry $D_{i,j}$

			B															
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
				A	G	T	C	G	C	C	G	C	T	G	C	T	G	C
R	0		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	1	A	1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	2	G	2	1	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	3	C	3	2	1	1	1	2	3	4	5	6	7	8	9	10	11	12
	4	G	4	3	2	2	2	1	2	3	4	5	6	7	8	9	10	11
	5	C	5	4	3	3	2	2	1	2	3	4	5	6	7	8	9	10
	6	T	6	5	4	3	3	3	2	2	3	4	4	5	6	7	8	9
	7	T	7	6	5	4	4	4	3	3	3	4	4	5	6	6	7	8
	8	G	8	7	6	5	5	4	4	4	3	4	5	4	5	6	6	7
	9	C	9	8	7	6	5	5	4	4	4	3	4	5	4	5	6	6
	10	T	10	9	8	7	6	6	5	5	5	4	3	4	5	4	5	6
	11	G	11	10	9	8	7	6	6	6	5	5	4	3	4	5	4	5
	12	C	12	11	10	9	8	7	6	6	6	5	5	4	3	4	5	4

Fig. 1. The $D_{i,j}$ matrix computed for the strings R (of length m = 12) and B (of length n = 15). The match ending at (12,12) = 3 on the *m'th* row thus yields the best alignment. Here $C_{CH} = C_{DE} = C_{IN} = 1$. Computation of this matrix uses $O(mn)$ time.

in the matrix is then the minimal cost of the editing operations required to transform $r_1 \cdots r_i$ to $b_1 \cdots b_j$ (see Fig. 1).

The following algorithm computes the matrix $D_{[0,\ldots,m;0,\ldots,n]}$

```
Initialization  $D_{0,0} := 0$.
                for all $j$, $1 \le j \le n$ , $D_{0,j} := j$ ,.
                for all $i$, $1 \le i \le m$ , $D_{i,0} := i$ ,.
for $i$:=1 to $m$ do
    for $j$:=1 to $n$ do
        $D_{i,j}$:= min ($D_{i-1,j} + C_{DE}$, $D_{i,j-1} + C_{IN}$, $D_{i-1,j-1}$ if $r_i = b_j$ or $D_{i-1,j-1} + C_{CH}$
        otherwise)
        ($D_{i,j}$ is the minimum of three numbers. These numbers are obtained
        from the predecessors of $D_{i,j}$ on its column, row and diagonal, respec-
        tively).
```

The computation will continue until $D_{m,n}$ has been reached. Filling the matrix would thus require $O(mn)$ time. In order to get the actual alignment, a backtracking procedure is carried out by following a minimizing path from $D_{m,n}$ to $D_{0,0}$. Variants of this basic method have been used by Needleman and Wunsch, Sankoff, Sellers, Goad and Kanehisa, Nussinov, Fickett and Wagner and Fisher (19).

3. An Improved Algorithm for Finding The Minimal Number of Differences Between Two Strings

Suppose we have two strings with k being the largest number of differences (insertions, deletions and changes) allowed between them. Obviously, this algorithm handles only cases where $n-m \leq k$. Namely, $R = r_1 \cdots r_m$ and $B = b_1 \cdots b_{m+k}$. In the previous section we showed an algorithm which finds their optimal alignment in $O(m^2)$ time. Here we give the improved algorithm which reaches the same result in $O(mk)$ time (18).

As we have noted in the previous section, computation of elements in the matrix which are far from the central diagonals are not going to figure in the final optimal alignment of the sequences. This suggests that a more efficient algorithm can be derived by limiting the computation and considering only the central diagonals.

For consecutive elements along a diagonal, the difference $D_{i,j} - D_{i-1,j-1}$ is either zero or one. This allows efficient storage of the information in the $D_{i,j}$ matrix. For each diagonal d of the matrix, it suffices to store the information which specifies the point $(i, j = i + d)$ on d where the $D_{i,j}$ values increase.

Let k be the maximal number of differences allowed between R and B. For a number of differences e $(e \leq k)$ and a diagonal d, let $L_{d,e}$ denote the largest row i such that $D_{i,i+d} = e$ ($D_{i,j}$ is on diagonal d). This definition implies that between the subpattern $r_1 \cdots r_{L_{d,e}}$ and the subtext $b_1 \cdots b_{L_{d,e}+d}$, there are e differences. (Note: $L_{d,e} + d$ is the corresponding j. $L_{d,e}$ is the maximal i that satisfies our requirements. The maximal $i + d$ yields the j parameter, since $j - i = d$). This definition also implies that $r_{L_{d,e}+1} \neq b_{L_{d,e}+d+1}$. Otherwise $L_{d,e} + 1$ rather than $L_{d,e}$ would have been the maximal row i with the same number of e differences.

If some $L_{d,e}$ with $e \leq k$ equals m, we can reach the end of the pattern (row number m in the matrix) without incurring more than k differences. In this case, and only in this case, the answer to the original question "does R occur in B- starting at b_1 - with at most k differences" is "yes". This knowledge of the $L_{d,e}$ suffices for our purpose.

A basic feature of the D matrix is that all elements $D_{i,i+d}$ $(D_{i,i-d})$ on the upper (lower) d'th diagonal necessarily exceed, or are equal to, d. Now, e in $L_{d,e}$ is just a particular entry along this diagonal. Hence, by definition, $|d| \leq e$. We also restrict our attention to $e \leq k$, i.e. at most k differences, and conclude therefore that $|d| \leq k$. We can thus focus on the central $2k + 1$ diagonals: the main diagonal, $i = j$ $(d = 0)$, the k diagonals above it, $d = 1, \ldots, k$ and the k

diagonals below it, $d = -1, \ldots, -k$. The time requirement for the computation is, then, $O(km)$, rather than $O(m^2)$ as in the case for the standard algorithm.

How do we compute $L_{d,e}$? Using the values of $L_{d,e-1}$, $L_{d-1,e-1}$ and $L_{d+1,e-1}$ a variable *row* is initialized and increased by one unit at a time till it hits the correct value of $L_{d,e}$.

The $O(km)$ algorithm is as follows:

```
1.  for d:=-(k+1) to (k+1) do
        $L_{d,|d|-2} := -\infty$;
        if $d < 0$
            then $L_{d,|d|-1} := |d| - 1$ ;
            else $L_{d,|d|-1} := -1$ ;
2. for e: = 0 to k  do
    for d: = -e to e  do

3.  $row = \max \begin{cases} L_{d,e-1} + 1 \\ L_{d-1,e-1} \\ L_{d+1,e-1} + 1 \end{cases}$
4.  while $r_{row+1} = b_{row+1+d}$ do
        $row := row + 1$
5.  $L_{d,e} = row$
6.  If $L_{d,e} = m$
    then print *yes* and Stop.
```

1. The values assigned in the first section of the initialization procedure have been selected in a way such that they will not interfere with the calculation of $L_{d,e}$'s.

2. The limits of the loops have already been explained: $e \leq k$ and $|d| \leq e$.

3. *row* is initialized by using three diagonals: d, $d - 1$ and $d + 1$. On each of these the largest row containing the $e - 1$ value is taken. We do not add one unit to the $L_{d-1,e-1}$ term since we are already on a lower diagonal and thus we move along the same row.

4. We continue on the same diagonal d as long as the nucleotide in the R sequence is the same as its counterpart on the B sequence and thus the number of differences e does not increase. When these two nucleotides differ, we set step 5. In step 6, the end of sequence R has been reached. Since $e \leq k$, an appropriate match has been found.

An example is given in Figure 2. Note, however, that it is not stored in the computer as such. A matrix is produced here only for the convenience of the reader. It suffices to store the necessary L's. This comment holds true for Figures 3a,b,c,d as well.

			B															
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
				A	G	T	C	G	C	C	G	C	T	G	C	T	G	C
	0		0	1	2	3												
	1	A	1	0	1	2	3											
	2	G	2	1	0	1	2	3										
	3	C	3	2	1	1	1	2	3									
	4	G		3	2	2	2	1	2	3								
	5	C			3	3	2	2	1	2	3							
	6	T				3	3	3	2	2	3							
R	7	T							3	3	3							
	8	G									3							
	9	C										3						
	10	T											3					
	11	G												3				
	12	C													3			

Fig. 2. Alignment of the B and R strings allowing at most k (here $k = 3$) differences. The $L_{d,e}$'s of this computation are:

$L_{0,-2} = -\infty$, $L_{0,-1} = -1$, $L_{0,0} = 2$, $L_{0,1} = 3$, $L_{0,2} = 6$, $L_{0,3} = 12$

$L_{1,-1} = -\infty$, $L_{1,0} = -1$, $L_{1,1} = 5$, $L_{1,2} = 6$, $L_{1,3} = 7$

$L_{2,0} = -\infty$, $L_{2,1} = -1$, $L_{2,2} = 5$, $L_{2,3} = 6$

$L_{3,1} = -\infty$, $L_{3,2} = -1$, $L_{3,3} = 5$

$L_{-1,-1} = -\infty$, $L_{-1,0} = 0$, $L_{-1,1} = 3$, $L_{-1,2} = 5$, $L_{-1,3} = 7$

$L_{-2,0} = -\infty$, $L_{-2,1} = 1$, $L_{-2,2} = 4$, $L_{-2,3} = 6$

$L_{-3,1} = -\infty$, $L_{-3,2} = 2$, $L_{-3,3} = 6$

The answer to the question "can we find an occurrence of string R with at most k differences starting at b_1 " is YES $L_{0,3} = 12$. The rightmost symbol is b_{12} and $S_{i,j}$ is: (0,0,2),(2,0,0),(3,2,3),(6,0,0),(7,7,5). This latter computation is explained in section 4 and is put in now so that this figure can also be used as the first stage in the text analysis algorithm. Now, we move the pattern to the right an lay it on the second symbol of the text. See Fig. 3a.

4. The Efficient String Matching Algorithm With k Differences

We next describe the new algorithm. First we build a table by analyzing the pattern. Then we examine the text from left to right checking possible occurrences with respect to one starting location (in the text) at each iteration. Besides the tables built in the pattern analysis, the input to each iteration consists of the knowledge acquired in previous iterations. The rightmost location in the text to which we have arrived in previous iterations is of particular significance. Each iteration consists of manipulating this knowledge. If neces-

				B													
				2	3	4	5	6	7	8	9	10	11	12	13	14	15
				G	T	C	G	C	C	G	C	T	G	C	T	G	C
	0		0	1	2	3											
	1	A	1	1	2	3											
	2	G	2	1	2	3	3										
	3	C	3	2	2	2	3	3									
	4	G		3	3	3	2	3									
	5	C				3	3	2	3								
	6	T						3	3								
R	7	T															
	8	G															
	9	C															
	10	T															
	11	G															
	12	C															

				B												
				3	4	5	6	7	8	9	10	11	12	13	14	15
				T	C	G	C	C	G	C	T	G	C	T	G	C
	0		0	1	2	3										
	1	A	1	1	2	3										
	2	G	2	2	2	2	3									
	3	C	3	3	2	3	2	3								
	4	G			3	2	3	3	3							
	5	C				3	2	3		3						
	6	T					3	3			3					
R	7	T														
	8	G														
	9	C														
	10	T														
	11	G														
	12	C														

				B											
				4	5	6	7	8	9	10	11	12	13	14	15
				C	G	C	C	G	C	T	G	C	T	G	C
	0		0	1	2	3									
	1	A	1	1	2	3									
	2	G	2	2	1	2	3								
	3	C	3	2	2	1	2	3							
	4	G		3	2	2	2	2	3						
	5	C			3	2	2	3	2	3					
	6	T				3	3	3	3	2	3				
R	7	T								3	3				
	8	G									3				
	9	C										3			
	10	T											3		
	11	G												3	
	12	C													3

Fig. 3a. This figure explains the $O(k^2n)$ text analysis algorithm. It is a continuation of Figure 2. Thus, here we put the pattern (R) on the text (B) at position b_2. The resulting calculation is:

$L_{0,-2} = -\infty$, $L_{0,-1} = -1$, $L_{0,0} = 0$, $L_{0,1} = 1$, $L_{0,2} = 5$, $L_{0,3} = 6$

$L_{1,-1} = -\infty$, $L_{1,0} = -1$, $L_{1,1} = 0$, $L_{1,2} = 1$, $L_{1,3} = 5$

$L_{2,0} = -\infty$, $L_{2,1} = -1$, $L_{2,2} = 0$, $L_{2,3} = 3$

$L_{3,1} = -\infty$, $L_{3,2} = -1$, $L_{3,3} = 0$

$L_{-1,-1} = -\infty$, $L_{-1,0} = 0$, $L_{-1,1} = 2$, $L_{-1,2} = 3$, $L_{-1,3} = 6$

$L_{-2,0} = -\infty$, $L_{-2,1} = 1$, $L_{-2,2} = 3$, $L_{-2,3} = 5$

$L_{-3,1} = -\infty$, $L_{-3,2} = 2$, $L_{-3,3} = 4$

How we compute $L_{(0,2)}$: $\mathit{row} = \mathit{max}(L_{-1,1}, L_{0,1}+1, L_{1,1}+1)$, $\mathit{row} = 2$. We compare now $b_4,\ldots$ to $r_3,\ldots$. From $S_{1,12}$ we get that $b_4b_5b_6 = r_3r_4r_5$. $f = 3$, $c = 2$. $g = \mathit{MAXLENGTH}(2,2) = 10$. Therefore, $L_{0,2} = \mathit{row} + \mathit{min}(f, g) = 2 + 3 = 5$.
How we compute $L_{1,3}$: $\mathit{row} = \mathit{max}(L_{0,2}, L_{1,2} + 1, L_{2,2} + 1)$, $\mathit{row} = 5$. We compare now $b_8,\ldots$ to $r_6,\ldots$. From $S_{1,12}$ we get that $b_8,\ldots,b_{12} = r_8,\ldots,r_{12}$. $f = 5$, $c = 7$. $g = \mathit{MAXLENGTH}(7,5)=0$. Therefore, $L_{0,2} = \mathit{row}+\mathit{min}(f, g) = 5 + 0 = 5$. The answer to the question posed in the caption of Fig. 2 is NO. None of the L's is equal to 12 (m). The rightmost symbol of the text which was checked is b_7. So, we do not create a new $S_{i,j}$ sequence.

Fig. 3b. The pattern is now laid starting at b_3. (See legend of Fig 3a). The calculation of the L's yields:

$L_{0,-2} = -\infty$, $L_{0,-1} = -1$, $L_{0,0} = 0$, $L_{0,1} = 1$, $L_{0,2} = 2$, $L_{0,3} = 5$

$L_{1,-1} = -\infty$, $L_{1,0} = -1$, $L_{1,1} = 0$, $L_{1,2} = 3$, $L_{1,3} = 4$

$L_{2,0} = -\infty$, $L_{2,1} = -1$, $L_{2,2} = 0$, $L_{2,3} = 6$

$L_{3,1} = -\infty$, $L_{3,2} = -1$, $L_{3,3} = 0$

$L_{-1,-1} = -\infty$, $L_{-1,0} = 0$, $L_{-1,1} = 1$, $L_{-1,2} = 5$, $L_{-1,3} = 6$

$L_{-2,0} = -\infty$, $L_{-2,1} = 1$, $L_{-2,2} = 2$, $L_{-2,3} = 6$

$L_{-3,1} = -\infty$, $L_{-3,2} = 2$, $L_{-3,3} = 3$

The answer to the question posed in the legend of Fig 2 is then NO. The rightmost symbol is b_{10}.

Fig. 3c. The pattern is now laid starting at b_4 (see legend to Fig 3a). The calculation of the L's yields:

$L_{0,-2} = -\infty$, $L_{0,-1} = -1$, $L_{0,0} = 0$, $L_{0,1} = 3$, $L_{0,2} = 4$, $L_{0,3} = 12$

$L_{1,-1} = -\infty$, $L_{1,0} = -1$, $L_{1,1} = 0$, $L_{1,2} = 6$, $L_{1,3} = 7$

$L_{2,0} = -\infty$, $L_{2,1} = -1$, $L_{2,2} = 0$, $L_{2,3} = 6$

$L_{3,1} = -\infty$, $L_{3,2} = -1$, $L_{3,3} = 0$

$L_{-1,-1} = -\infty$, $L_{-1,0} = 0$, $L_{-1,1} = 1$, $L_{-1,2} = 5$, $L_{-1,3} = 6$

$L_{-2,0} = -\infty$, $L_{-2,1} = 1$, $L_{-2,2} = 5$, $L_{-2,3} = 6$

$L_{-3,1} = -\infty$, $L_{-3,2} = 2$, $L_{-3,3} = 6$

How we compute $L_{0,3}$: $\mathit{row} = \mathit{max}(L_{-1,2}$, $L_{0,2} + 1$, $L_{1,2} + 1)$, $\mathit{row} = 7$. We compare now $b_{11},\ldots$ to $r_8,\ldots$. From $S_{1,12}$ we get that $b_{11}b_{12} = r_{11}r_{12}$. $f = 2$, $c = 10$. $g = \mathit{MAXLENGTH}(10,7) = 2$. $f = g = 2$. Therefore $\mathit{row} = \mathit{row} + f = 9$ and we compare again. Since, here we have no information from $S_{1,12}$, we check symbol by symbol. We find that $b_{13}b_{14}b_{15}$ is equal $r_{10}r_{11}r_{12}$ and $\mathit{row} = 12$. The answer to the question posed in the caption to Fig. 2 is YES $L_{0,3} = 12$. The rightmost symbol is b_{15} and the new $S_{i,j}$ $(S_{3,15})$ is: (3,0,0),(4,1,2),(6,0,0),(7,3,3),(10,7,5).

MAXLENGTH

		A	G	C	G	C	T	T	G	C	T	G	C
		0	1	2	3	4	5	6	7	8	9	10	11
A	0	12	0	0	0	0	0	0	0	0	0	0	0
G	1	0	11	0	2	0	0	0	2	0	0	2	0
C	2	0	0	10	0	1	0	0	0	1	0	0	1
G	3	0	2	0	9	0	0	0	3	0	0	2	0
C	4	0	0	1	0	8	0	0	0	2	0	0	1
T	5	0	0	0	0	0	7	1	0	0	1	0	0
T	6	0	0	0	0	0	1	6	0	0	3	0	0
G	7	0	2	0	3	0	0	0	5	0	0	2	0
C	8	0	0	1	0	2	0	0	0	4	0	0	1
T	9	0	0	0	0	0	1	3	0	0	3	0	0
G	10	0	2	0	2	0	0	0	2	0	0	2	0
C	11	0	0	1	0	1	0	0	0	1	0	0	1

Fig. 4. Computation of *MAXLENGTH*. Here, we describe the pattern analysis. The pattern is an array $R = r_1, \ldots, r_m$. The output of the pattern analysis is the two dimensional array *MAXLENGTH*[0,...,$m-1$;0,...,$m-1$] where *MAXLENGTH*$(i,j) = f$ means that $r_{i+1}, \ldots, r_{i+f} = r_{j+1}, \ldots, r_{j+f}$ and $r_{i+f+1} \neq r_{j+f+1}$.
The array *MAXLENGTH* is symmetric. That is, *MAXLENGTH*(i,j)= *MAXLENGTH*(j,i). We will apply a slight modification of the string matching algorithm of (20) due to (21). Given a pattern of length m and a text of length n this modification finds for each entry of text one of two things:
1. Whether an occurrence of the pattern starts at this entry.
2. The first mismatch (from the left). That is, it finds the first character of the text which differs from a character of the pattern.
We separately compute each row i of *MAXLENGTH*. Take $r_{i+1}, \ldots, r_m$ to be the pattern and $r_1, \ldots, r_m$ to be the text. We compute *MAXLENGTH*[i;1,...,$m-1$] simply by applying this modification. The computation of each row takes $O(m)$ time. Since, There are m rows the total time is $O(m^2)$.

sary, we proceed to investigate the text to the right of this rightmost location. (As examples use figures 2, 3a, b, c, d).

The first part of the algorithm is the pattern analysis. The output of the pattern analysis is a two dimensional array *MAXLENGTH*[0,...,$m-1$;0,...,$m-1$]. *MAXLENGTH*$(i,j) = f$ means that $r_{i+1}, \ldots, r_{r+f} = r_{j+1}, \ldots, r_{j+f}$, and $r_{i+f+1} \neq r_{j+f+1}$. This means that if we lay a section of the pattern starting at r_{i+1} over another section, starting at r_{j+1}, f is the largest exact repeat of the two sections. This exact repeat will end at r_{i+f} in the first section and at r_{j+f} in the second. The pattern analysis takes $O(m^2)$ time. It is exceedingly simple. Fig 4 presents an actual example and its caption gives a brief description of the computation.

Let us now return to the text analysis which consists of $n-m+k+1$ iterations (i.e from 0 to $n-m+k$). At iteration i we check for an occurrence with $\leq k$ differences of the pattern starting at b_{i+1}. Let b_j be the rightmost nucleotide in the text that was reached first at some prior iteration, say the l th ($0 \leq l < i$). Since we look for matches with at most k differences between the pattern and some section of the text, there are necessarily $\leq k$ differences between $b_{i+1}, \ldots, b_j$ and the portion of the pattern matched with it.

One example is repeatedly given in the figures (2, 3a, b, c, d). Let us here give an additional example. Let $r_1, \ldots, r_{13}$ be *ACTACTTTCCGAG* and let $b_{17}, \ldots, b_{30}$ be *AGCTACTTGTCCAG* (with iteration l = 16 , j = 30). The correspondence is:

```
     1       2  3  4   5  6  7     8  9  10 11 12 13
R |  A       C  T  A   C  T  T     T  C  C  G  A  G
     |       |  |  |   |  |  |     |  |  |     |  |
B |  A   G   C  T  A   C  T  T  G  T  C  C     A  G
     17  18  19 20 21  22 23 24 25 26 27 28    29 30
     b_{l+1}       b_{i+1}                        b_j
```

The correspondence gives k = 3 differences. Hence there are also $\leq k$ differences between some portion of the pattern and $b_{i+1}, \ldots, b_j$ (i = 20). Let us call this, $r_4, \ldots, r_{13}$, portion the subpattern. Thus, for some correspondence between the nucleotides of $b_{i+1}, \ldots, b_j$ (i. e. $b_{21}, \ldots, b_{30}$) and the subpattern, there are at least $j-i-k$ (30−20−3) matched nucleotides. It is easy to see that all the nucleotides of the text that have successive matches with the subpattern form at most $k+1$ successive substrings. For each substring of the text we know its corresponding substring in the pattern. When a substring of the text $b_{p+1}, \ldots, b_{p+f}$ matches a substring of the pattern $r_{c+1}, \ldots, r_{c+f}$ (e.g. p = 18 , f = 6 , and c = 1 , $r_{1+1}, \ldots, r_{1+6} = b_{18+1}, \ldots, b_{18+6}$) and $b_{p+f+1} \neq r_{c+f+1}$, denote this by the ***triple*** (p, c, f) (e.g. (18,1,6)). In the section $b_{i+1}, \ldots, b_j$ there are at most k unmatched and/or mismatched nucleotides. Each of these b_h nucleotides is denoted by the triple $(h, 0, 0)$ (in our example h = 17 , $b_{h+1} = b_{18} = G$).

Thus, the substring of interest in the text, $b_{i+1}, \ldots, b_j$, ($b_{21}, \ldots, b_{30}$) breaks up into $O(k)$ consecutive matched or unmatched elements as follows: (20,3,4),(24,0,0),(25,7,3),(28,11,2). These triples are denoted as $S_{i,j}$ ($S_{20,30}$).

In order to find the best alignment during iteration i we always use the algorithm described in section 3 to guide us. In addition, however we use the output of *MAXLENGTH* (pattern analysis) and $S_{i,j}$, the output of a previous iteration.

In iteration i the pattern $r_1, \ldots, r_m$ is put on $b_{i+1}, \ldots,$. Returning to our example, for i = 20 we have:

					1	2	3	4	5	6	7	8	9	10	11	12	13	
R |					A	C	T	A	C	T	T	T	C	C	G	A	G	
B |	A	G	C	T	A	C	T	T	G	T	C	C	A	G	A	A	G	...
					21	22	23	24	25	26	27	28	29	30	31	32	33	...
					b_{i+1}									b_j				

Iteration i consists of the $O(km)$ algorithm of section 3. However, it uses a modification of the instruction 4 presented there. Instruction 4 increases the variable row by one unit at a time. The availability of the sequence of triples $S_{i,j}$ and *MAXLENGTH* allows increasing row by much larger jumps as long as we do not require information about symbols of the text, which are beyond b_j. Once row takes us beyond b_j, $S_{i,j}$ and *MAXLENGTH* do not help us any more and we apply (the old) instruction 4 as in the computation of the previous algorithm.

Let us now explain instruction 4.new. The *while* loop of the new instruction 4 looks for the longest exact match w between any section of the text $b_{i+row+d+1}$ and some portion of the pattern r_{row+1} (as long as $i+1 \le i+row+d+1 \le j$). According to $S_{i,j}$ (i.e. the listing of triples) $b_{i+row+d+1}, \dots, b_{i+row+d+f}$ matches $r_{c+1}, \dots, r_{c+f}$ for some index c of the pattern. In the computation of w we have the following cases:

Case (a). $f \ge 1$. In our example (i =20), assume that row = 8 and d =−2 and we want to find what is the longest exact match, w, between $b_{27}, \dots,$ and $r_9, \dots,$. Inspecting $S_{20,30}$ we find the triple (25,7,3), meaning that nucleotides 27 to 28 of the text match nucleotides 9 to 10 of the pattern (f = 2 and c = 8). Thus, this triple *covers* nucleotide 27. At this stage we refer to *MAXLENGTH*. *MAXLENGTH*(c,row) gives .the maximal number g such that $a_{c+1}, \dots, a_{c+g}$ equals $a_{row+1}, \dots, a_{row+g}$. In our example $r_{8+1}, \dots, r_{8+5} = r_{8+1}, \dots, r_{8+5}$. g equals then to 5. Case (a) has two subcases:

Case (a1). $f \ne g$. It is easy to see that here $b_{i+row+d+1}, \dots, b_{i+row+d+min(f,g)} = r_{row+1}, \dots, r_{row+min(f,g)}$ and $(b_{i+row+d+min(f,g)+1} \ne r_{row+min(f,g)+1})$. The longest match w here is min(f,g). Therefore, we assign $w := min(f,g)$, and row increases by w. In our example w = 2, (min(2,5)), and row = row + 2 = 10.

Case (a2). $f = g$. This implies $b_{i+row+d+1}, \dots, b_{i+row+d+f} = r_{row+1}, \dots, r_{row+f}$ but does not reveal whether the next nucleotide in the text, namely $b_{i+row+f+1}$ is equal to the next nucleotide in the pattern $r_{row+f+1}$. Therefore, we assign $row := row + f$ and apply again the present case analysis accumulating the "jump" over f into w.

Case (b). f = 0, namely $b_{i+row+d+1}$ is unmatched in the $S_{i,j}$ triples. In this case we compare nucleotide $b_{i+row+d+1}$ to its corresponding one in the pattern, r_{row+1}. For example, compare b_{25} to r_4, (i = 20 , row = 3 , d = 1). This case has two subcases.

Case (b 1). The two nucleotides differ. In this case $w := 0$.

Case (b2). The two nucleotides are identical. In this case $w = 1$ and we assign $row := row + 1$, and apply again the present case analysis accumulating this propagation of 1 into w. Detailed examples are given in figures 2,3a,b,c,d.

The text analysis algorithm is

```
j:=0;
for i:=0 to n-m+k do
begin
    [1] Proper initialization (as in instruction 1 of the O(km) algorithm)
    [2] for e:=0 to k do
        for d:=-e to e do
        begin
            [3] row := max[($L_{d,e-1}$+1),($L_{d-1,e-1}$),($L_{d+1,e-1}$+1)].
            [4.new] while i+row+d+1 ≤ j do
                [4.new.1] take from $S_{i,j}$ the triple that "covers" $b_{i+row+d+1}$.
                Derive from this triple the indices c,f such that
                $b_{i+row+d+1}, \dots, b_{i+row+d+f} = r_{c+1}, \dots, r_{c+f}$
                [4.new.2] if f ≥ 1
                then (* case a *)
                    [4.new.3] if f ≠ MAXLENGTH(c,row)
                    then (* case a1 *)
                        row := row + min(f,MAXLENGTH(c,row))
                        go to 5
                    else (* case a2 *)
                        row:= row + f ;
                else (* case b *)
                    [4.new.4] if $b_{i+row+d+1} \neq r_{row+1}$
                    then (* case b1 *)
                        go to 5
                    else (* case b2 *)
                        row:= row+1
            od
            [4.old] while $r_{row+1} = b_{i+row+1+d}$ do
                row:=row+1.
            [5] $L_{d,e}$:= row.
            [6] if $L_{d,e}$ = m
            then print *YES* and go to 7
        end
    [7] If new symbols of the text were reached (j was increased), then starting
    from the $L_{d,k}$ (which implies the new j, i.e. j=$L_{d,k}$ + d + i) we reconstruct
    the new $S_{i,j}$.
end
```

Implementation remarks.

Instruction 4.new.1: When we compute $L_{0,0}$ we start searching for the indices (f,c) at the first triple of $S_{i,j}$. We know which triple was checked when any $L_{d,e}$ gets its value. So, when computing a new $L_{d,e}$, we already know which were the last triples checked in the computation of each one of $L_{d-1,e-1}$,$L_{d,e-1}$ and $L_{d+1,e-1}$. In instruction 3 *row* got its initial value from the maximum of $L_{d-1,e-1}, L_{d,e-1}+1$ and $L_{d+1,e-1}+1$. The last triple checked in the computation of the maximum is the first to be considered now, in the computation of $L_{d,e}$.

Instruction 7: If at least one new symbol of the text is reached, at the end of iteration i a new sequence of triples is created instead of $S_{l,j}$. If $b_{\bar{j}}$ is the rightmost symbol of the text reached in such an iteration (i), then denote the new sequence $S_{i,\bar{j}}$. For each $L_{d,e}$ computed during iteration i. we keep a sequence (of triples). This sequence "realizes" $L_{d,e}$. That is, it gives a correspondence between $r_1, \ldots, r_{L_{d,e}}$ and $b_{i+1}, \ldots, b_{i+L_{d,e}+d}$, with exactly e differences. This sequence is used in the computation of $S_{i,\bar{j}}$.

At the beginning of each iteration i, each $L_{d,e}$ has an empty triple sequence. Here, we use again the fact that initially (at instruction 3) *row* is the maximum among $L_{d-1,e-1}$, $L_{d,e-1}+1$, and $L_{d+1,e-1}+1$ and finally (at instruction 5) is $L_{d,e}$. Assume that we know the sequences of the predecessors of $L_{d,e}$ (namely, the sequences of $L_{d-1,e-1}$, $L_{d,e-1}$ and $L_{d+1,e-1}$). We get the sequence of $L_{d,e}$ by adding triples to the end of the sequence of the predecessor which gives the maximum in initializing *row*. Let l_1 be the initial value of *row*. If l_1 got its value from $L_{d-1,e-1}$ (or $L_{d,e-1}$) then we add to its sequence the triple $(i+l_1+d-1, 0, 0)$. (Meaning that for b_{i+l_1+d}, there is no corresponding symbol in the pattern). Following instruction 5, if $L_{d,e} > l_1$, we next add the triple $(i+l_1+d, l_1, L_{d,e}-l_1)$ to the sequence of $L_{d,e}$. This last triple describes the match between substrings of the pattern and the text. It was found during the computation of $L_{d,e}$ given $L_{d-1,e-1}$, $L_{d,e-1}$ and $L_{d+1,e-1}$. This latter addition is done regardless of whether the source of l_1 was $L_{d-1,e-1}$, $L_{d,e-1}$ or $L_{d+1,e-1}$.

At the end of iteration i we check which of the sequences of the $2k+1$ $L_{d,k}$'s reach the rightmost symbol of the text. If the index of this symbol is greater than j ($L_{d,e}+d+i > j$), we take its sequence to be the new $S_{i,j}$.

The old instruction 4 (where *row* is increased by one unit at a time without using $S_{i,j}$ and *MAXLENGTH*) is employed each time we move to a new symbol of the text. We maintain $O(k)$ diagonals at any time during the text analysis and may need to compare the new symbol for each of them. Hence, the old instruction 4 requires a total of $O(kn)$ time throughout the text analysis. In order to evaluate the number of steps which are required by the new instruction 4 at iteration i, we use again the fact that $O(k)$ diagonals are computed. The sequence $S_{i,j}$ has at most $2k+1$ triples. We can charge each operation performed on any one of the diagonals to either a difference being discovered (there are $\leq k$ such differences), or to a triple of $S_{i,j}$ being examined (there are $\leq 2k+1$ triples). This amounts to $O(k)$ operations per diagonal at each iteration

i. Therefore, the total running time of the text analysis is $O(k^2n)$. This implies that the program will run the fastest when only good alignments are accepted (i.e. k is stipulated to be small). The actual alignment of the pattern and text can easily be produced if we retain all those $S_{i,j}$ where the end of the pattern (m) has been reached. Along with each $S_{i,j}$, its corresponding number of differences ($\leq k$) can be stored.

A preliminary version of the program has been written in Fortran and implemented on the CDC 6600. This program has been used to calculate the figures given here.

5. Discussion

This paper constitutes an additional example of a successful interdisciplinary effort where an algorithm developed by computer science designers is applied to solve problems that arise in molecular biology. The principle of the algorithm we have described, though seemingly complex, is essentially simple.

As we slide the pattern along the text, we shall then get several alignments, each with at most k differences, rather than only the position in the text of the optimal alignment. Clearly this is advantageous both in comparing amino acid and nucleotide sequences. One may then list all alignments either by site or by their minimal differences score, i.e. by how good they are.

For the problem of efficient string matching in the presence insertions and deletions, the algorithms presented here are the most efficient to date. Especially noteworthy is the algorithm described in section 4. Whenever the text analysis (rather than the pattern) dominates the computation time, the running time of this algorithm is close to optimal.

Very often long sequences are involved in homology searches. Also very frequent is the need to compare one sequence (i.e. the pattern) to many others.

Thus, the fast growing rate in the sequencing of DNA (and protein) molecules, coupled with the efficiency of the algorithm, as well as the listing it provides of all alignments better than a previously set value, make it an attractive tool in molecular biology.

ACKNOWLEDGEMENTS U. V. has been supported by NSF grant NSF-DCR-8318874 and ONR grant N0014-85-K-0046.

*To whom correspondence should be addressed

REFERENCES

1. Landau, G. M. and Vishkin, U. (1985). Proc. 26th IEEE Symp. on Foundation of Comp. Sci. In press.
2. Landau, G. M. and Vishkin, U. (1985). Submitted.
3. Tinoco, I. Jr. Uhlenbeck, O. C. and Levine, M. D. (1971). Nature 230, 362-367.
4. Maizel, J. V. Jr. and Lenk, R. P. (1981). Proc. Natl. Acad. Sci. USA 78, 7665-7669.
5. Needleman, S. B. and Wunsch, C. D. (1970). J. Mol. Biol. 48, 443-453.
6. Sankoff, D. (1972). Proc. Natl. Acad. Sci. USA 69, 4-6.
7. Sellers, P. H. (1974). SIAM J. Appl. Math. 26, 787-796.
8. Dumas, J. -P. and Ninio, J. (1982). Nucl. Acids Res. 10,197-206.
9. Korn, L. J., Queen, C. L. and Hegman, M. N. (1977). Proc. Natl. Acad. Sci. USA 74, 4401-4405.
10. Sellers, P. H. (1979). Proc. Natl. Acad. Sci. USA 76, 3041.
11. Sellers, P. H. (1980). J. Algorithms 1, 359-373.
12. Goad, W. B. and Kanehisa, M. I. (1982). Nucl. Acids Res. 10, 247-263.
13. Wilbur, W. J. and Lipman, D. (1983). Proc. Natl. Acad. Sci. USA 80, 726-730.
14. Nussinov, R. (1983). J. Theor. Biol. 100, 319-328.
15 Nussinov, R. , Pieczenik, G. , Griggs, J. R. and Kleitman, D. J. (1978). SIAM J. Appl. Math. 35 (1) 68-82.
16 Nussinov, R. and Jacobson, A. B. (1980). Proc. Natl. Acad. Sci. USA 77, 6309-6313.
17. Fickett, J. W. (1984). Nucl. Acids Res. 12, 175-180.
18. Ukkonen, E. (1983). Proc. Inter. Conf. Found. Comp. Theor. Lecture Notes Comp. Sci 158, 487-495.
19. Wagner, R. and Fisher, M. (1974). J. ACM 21, 168-178.
20. Knuth, D. E., Morris, J. H. and Pratt, V. R. (1977). SIAM J. Comp. 6, 322-350.
21. Main M. G. and Lorentz, R. J. (1984). J. of Algorithms 5, 422-432.